VSB TECHNICAL UNIVERSITY OF OSTRAVA | FACULTY OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE | DEPARTMENT OF COMPUTER SCIENCE

Transform and Conquer

Jiří Dvorský, Ph.D.

Presentation status to date February 24, 2025

Department of Computer Science
VSB – Technical University of Ostrava

Transform and Conquer

Presorting

- Unity of elements in the array
- Module Calculation
- Search

Gaussian Elimination Method

- LU-decomposition of a matrix

Balanced Search Trees

- AVL Trees
- 2-3 trees

Lecture outline (cont.)

Heap and Heap Sorting

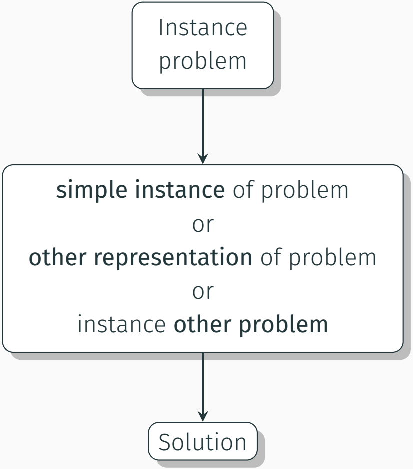
Horner's Scheme

Problem Reduction

Solution strategy transform and solve

Biphasic strategy

1. transformation
2. solution



Transform and Conquer

Presorting

- A relatively old idea that motivated, among other things, research into sorting algorithms.
- Sorted data lead to significantly simpler algorithms, “order must be”.
- Prerequisites:
    1. data is stored in an array – sorting an array is easier than sorting a list **for s do**

I O

end
rting we use an algorithm with complexity $\Theta(n \log n)$ – typically QuickSort, MergeSort.

- Usage: geometric algorithms, graph algorithms, caustic algorithms.

Background

We are given an array A with n elements. We have to determine whether each element occurs exactly once in the array A.

Rough force solution – compare all pairs of elements until:

1. does not find a pair of the same elements or
2. tested all pairs of elements.

The time complexity is in the worst case $\Theta(n^2)$.

Unity of elements in the array

ALGORITHM *PresortElementUniqueness*$(A[0..n-1])$

//Solves the element uniqueness problem by sorting the array first
//Input: An array $A[0..n-1]$ of orderable elements
//Output: Returns "true" if A has no equal elements, "false" otherwise
sort the array A
for $i \leftarrow 0$ **to** $n-2$ **do**
    if $A[i] = A[i+1]$ **return false**
return true

Algorithm time complexity

$$T(n) = T_{sort}(n) + T_{scan}(n) \in \Theta(n \log n) + \Theta(n) = \Theta(n \log n)$$

Module count

Background

We are given an array A with n elements. We have to determine which element occurs most often in the array. This element is called **modus**.

For simplicity, we will assume that there is only one modus in the array A.

Rough force solution

For each element $a_i \in A$, search the auxiliary list L:

1. If we find a match, we increment the corresponding frequency,
2. otherwise, insert the element a_i at the end of the list with frequency 1.

Mod calculation – time complexity of brute force solution

- Worst case – all elements in array A are different.
- For a_i we have to do $i - 1$ comparison with elements in the list L before we add a new element to the end of it.
- The number of comparisons is equal to

$$C(n) = \sum_{i=1}^{n}(i-1) = 0 + 1 + \dots + (n-1) = \frac{1}{2}n(n-1) \in \Theta(n^2)$$

- Finding the maximum requires $n - 1$ comparisons, which does not affect the quadratic complexity of the algorithm.

Mod calculation – data presort

- If we sort the array A, the identical elements in the array A will be next to each other.
- To calculate the mode, it is enough to find the longest run of identical elements in A.
- Time complexity

$$T(n) = T_{sort}(n) + T_{scan}(n) \in \Theta(n \log n) + \Theta(n) = \Theta(n \log n)$$

ALGORITHM *PresortMode*($A[0..n-1]$)

```
//Computes the mode of an array by sorting it first
//Input: An array A[0..n − 1] of orderable elements
//Output: The array's mode
sort the array A
i ← 0                          //current run begins at position i
modefrequency ← 0              //highest frequency seen so far
while i ≤ n − 1 do
    runlength ← 1;   runvalue ← A[i]
    while i + runlength ≤ n − 1 and A[i + runlength] = runvalue
        runlength ← runlength + 1
    if runlength > modefrequency
        modefrequency ← runlength;   modevalue ← runvalue
    i ← i + runlength
return modevalue
```

Search for element x in array A of length n

- The brute force solution leads to an algorithm requiring n comparisons in the worst case.
- After sorting the array, the interval halving algorithm can be used, which requires $\lfloor \log_2 n \rfloor + 1$ comparison in the worst case.
- The time complexity of the algorithm will then be

  $$T(n) = T_{sort}(n) + T_{search}(n) = \Theta(n \log n) + \Theta(\log n) = \Theta(n \log n),$$

  which is **more** than the complexity of sequential search!!!
- But for **repeated** searches it is already worth sorting the A field.

Resources for self-study

- Book [1], chapter 6.1, pages 202 – 205

Transform and Conquer

Gaussian Elimination Method

Gaussian Elimination Method – Motivation

A system of two equations with two unknowns

$$\begin{aligned} a_{11}x + a_{12}y &= b_1 \\ a_{21}x + a_{22}y &= b_2 \end{aligned}$$

can be solved relatively easily – for example, we can express the variable x as a function of y, substitute it into the second equation, and solve the equation.

Problem

How to solve a system of n equations with n unknowns? In the same way?

Gaussian elimination method

System of n linear equations with n unknowns

$$\begin{aligned}
a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 \\
a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2 \\
&\vdots \\
a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n &= b_n
\end{aligned}$$

is transformed into an equivalent system of equations, where all coefficients below the main diagonal are zero

$$\begin{aligned}
a'_{11}x_1 + a'_{12}x_2 + \cdots + a'_{1n}x_n &= b'_1 \\
a'_{22}x_2 + \cdots + a'_{2n}x_n &= b'_2 \\
&\vdots \\
a'_{nn}x_n &= b'_n
\end{aligned}$$

Gaussian Elimination Method – Matrix Notation

$$\mathbf{A}\vec{x} = \vec{b} \quad \Longrightarrow \quad \mathbf{A}'\vec{x} = \vec{b}'$$

where

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & & & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix} \qquad \vec{b} = \begin{pmatrix} b_{11} \\ b_{21} \\ \vdots \\ b_{n1} \end{pmatrix}$$

$$\mathbf{A}' = \begin{pmatrix} a'_{11} & a_{12} & \cdots & a'_{1n} \\ 0 & a'_{22} & \cdots & a'_{2n} \\ \vdots & & & \vdots \\ 0 & 0 & \cdots & a'_{nn} \end{pmatrix} \qquad \vec{b}' = \begin{pmatrix} b'_{11} \\ b'_{21} \\ \vdots \\ b'_{n1} \end{pmatrix}$$

$\mathbf{A}'$ is called the upper triangular matrix.

Gaussian Elimination Method – Advantages of Representation Change

A system given by an upper triangular matrix can be easily solved using **back substitution**:

1. From the equation
$$a'_{nn} x_n = b'_n$$
we compute the unknown x_n.
2. We substitute the value of the unknown x_n into the equation
$$a'_{n-1\,n-1} x_{n-1} + a'_{n-1\,n} x_n = b'_{n-1}$$
and compute the unknown x_{n-1}.
3. We proceed in this manner until we compute the unknown x_1.

The complexity of this algorithm is $\Theta(n^2)$.

Gaussian elimination method – elementary operations

The matrix of the system $\mathbf{A}$ is transformed into an upper triangular matrix $\mathbf{A}'$ using **elementary operations**:

- swapping two equations in the system,
- multiplying an equation by a non-zero coefficient and
- adding or subtracting a multiple of another equation to the given equation, i.e. a linear combination with another equation.

Elementary operations do not change the solution of the system of equations – the transformed system has the same solution as the original system.

Gaussian Elimination Method – Matrix Transformation

1. We choose a_{11} as the **pivot** and "nullify" all coefficients in the first column, except for a_{11}.

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & & & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix}$$

   "Nullification" – from the second equation, we subtract $\frac{a_{21}}{a_{11}}$ times the first equation, from the third equation, we subtract $\frac{a_{31}}{a_{11}}$ times the first equation, and so on.
2. We choose a_{22} as the pivot and repeat the same procedure.

Remark

Of course, we also perform changes for the vector of right-hand sides $\vec{b}$.

Gaussian Elimination Method – Example

Let us have a system of equations

$$\begin{aligned} 2x_1 - x_2 + x_3 &= 1 \\ 4x_1 + x_2 - x_3 &= 5 \\ x_1 + x_2 + x_3 &= 0 \end{aligned}$$

The augmented matrix of the system

$$\left(\begin{array}{ccc|c} 2 & -1 & 1 & 1 \\ 4 & 1 & -1 & 5 \\ 1 & 1 & 1 & 0 \end{array}\right)$$

Gaussian Elimination Method – Example (cont.)

Forward Elimination

From the second row, we subtract $\frac{4}{2}$ times the first row, from the third row, we subtract $\frac{1}{2}$ times the first row

$$\left(\begin{array}{ccc|c} 2 & -1 & 1 & 1 \\ 0 & 3 & -3 & 3 \\ 0 & \frac{3}{2} & \frac{1}{2} & -\frac{3}{2} \end{array}\right)$$

From the third row, we subtract $\frac{\frac{3}{2}}{3} = \frac{1}{2}$ times the second row

$$\left(\begin{array}{ccc|c} 2 & -1 & 1 & 1 \\ 0 & 3 & -3 & 3 \\ 0 & 0 & 2 & -2 \end{array}\right)$$

Gaussian Elimination Method – Example (cont.)

Back Substitution

$$
\begin{aligned}
x_3 &= \frac{-2}{2} = -1 \\
x_2 &= \frac{3-(-3)x_3}{3} = \frac{3-(-3)(-1)}{3} = 0 \\
x_1 &= \frac{1-x_3-(-1)x_2}{2} = \frac{1-(-1)}{2} = 1
\end{aligned}
$$

Gaussian elimination method – forward elimination

Input : Matrix $\mathbf{A}$ of type $n \times n$ and column vector $\vec{b}$ of dimension n

Output: Equivalent triangular matrix $\mathbf{A}$ and vector $\vec{b}$

```
for i ← 1 to n − 1 do
    for j ← i + 1 to n do
        temp ← A[j, i]/A[i, i];
        for k ← i to n do
            A[j, k] ← A[j, k] − A[i, k] ∗ temp;
        end
        b[j] ← b[j] − b[i] ∗ temp;
    end
end
```

Partial Pivoting

- In the forward elimination algorithm, there is an error. If $a_{ii} = 0$, then division by zero occurs.
- The problem can be solved by swapping equations (elementary operation) so that $a_{ii} \neq 0$.
- It is also possible to simultaneously address potential rounding errors – the pivot is chosen such that it is the largest of all elements a_{ii} to a_{ni} in absolute value.

ALGORITHM *BetterForwardElimination*$(A[1..n, 1..n], b[1..n])$

```
//Implements Gaussian elimination with partial pivoting
//Input: Matrix A[1..n, 1..n] and column-vector b[1..n]
//Output: An equivalent upper-triangular matrix in place of A and the
//corresponding right-hand side values in place of the (n + 1)st column
for i ← 1 to n do A[i, n + 1] ← b[i] //appends b to A as the last column
for i ← 1 to n − 1 do
    pivotrow ← i
    for j ← i + 1 to n do
        if |A[j, i]| > |A[pivotrow, i]|  pivotrow ← j
    for k ← i to n + 1 do
        swap(A[i, k], A[pivotrow, k])
    for j ← i + 1 to n do
        temp ← A[j, i] / A[i, i]
        for k ← i to n + 1 do
            A[j, k] ← A[j, k] − A[i, k] ∗ temp
```

Gaussian Elimination Method – Time Complexity

- Input size – number of equations in the system, i.e., dimension of matrix n.
- Basic operation – arithmetic operations, for historical reasons multiplication. In the innermost cycle, the number of multiplications corresponds to the number of subtractions, it's just a multiple of a constant 2.
- We will be interested in the number of multiplications $C(n)$ depending on the number n.

Gaussian Elimination Method – Time Complexity (cont.)

$$
\begin{aligned}
C(n) &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \sum_{k=i}^{n} 1 = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} (n - i + 1) \\
&= \sum_{i=1}^{n-1} (n - i + 1) \sum_{j=i+1}^{n} 1 = \sum_{i=1}^{n-1} (n - i + 1)(n - i)
\end{aligned}
$$

The last sum is expanded for individual i

$$
\begin{array}{cccc}
i = 1 & (n - 1 + 1)(n - 1) & = & n(n - 1) \\
i = 2 & (n - 2 + 1)(n - 2) & = & (n - 1)(n - 2) \\
\vdots & \vdots & \vdots & \vdots \\
i = n - 2 & (n - n + 2 + 1)(n - n + 2) & = & 3 \cdot 2 \\
i = n - 1 & (n - n + 1 + 1)(n - n + 1) & = & 2 \cdot 1
\end{array}
$$

Gaussian Elimination Method – Time Complexity (cont.)

From the last column, it is clear that this is a sum of a series

$$1 \cdot 2 + 2 \cdot 3 + \cdots + (n-2)(n-1) + (n-1)n = \sum_{l=1}^{n-1} l(l+1)$$

$$\begin{aligned}
\sum_{l=1}^{n-1} l(l+1) &= \sum_{l=1}^{n-1} l^2 + \sum_{l=1}^{n-1} l \\
&= \frac{1}{6}n(n-1)(2n-1) + \frac{1}{2}n(n-1) \\
&= \frac{1}{3}n^3 - \frac{1}{2}n^2 + \frac{1}{6}n + \frac{1}{2}n^2 - \frac{1}{2}n \\
&= \frac{1}{3}n^3 - \frac{1}{3}n
\end{aligned}$$

Gaussian Elimination Method – Time Complexity (cont.)

And therefore

$$C(n) = \frac{1}{3}n^3 - \frac{1}{3}n \approx \frac{1}{3}n^3 \in \Theta(n^3)$$

Since the complexity of back substitution is $\Theta(n^2)$, the complexity of the entire Gaussian elimination method is $\Theta(n^3)$.

LU-decomposition of a matrix

Let us have the matrix $\mathbf{A}$ of the system of linear equations from the previous example

$$\mathbf{A} = \begin{pmatrix} 2 & -1 & 1 \\ 4 & 1 & -1 \\ 1 & 1 & 1 \end{pmatrix}$$

Further, let us consider two matrices:

$$\mathbf{L} = \begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ \frac{1}{2} & \frac{1}{2} & 1 \end{pmatrix}$$

Coefficients from Gaussian elimination

$$\mathbf{U} = \begin{pmatrix} 2 & -1 & 1 \\ 0 & 3 & -3 \\ 0 & 0 & 2 \end{pmatrix}$$

Result of Gaussian elimination

LU-decomposition of a matrix

Definition

Let $\mathbf{A}$ be a regular square matrix with elements from the real numbers, for which it is not necessary to swap rows during Gaussian elimination. Then there exist regular matrices $\mathbf{L}$ and $\mathbf{U}$, which are uniquely determined and satisfy the following statement

$$\mathbf{A} = \mathbf{L}\mathbf{U},$$

where $\mathbf{L}$ is a lower triangular matrix with ones on the entire main diagonal and $\mathbf{U}$ is an upper triangular matrix with non-zero elements on the main diagonal.

Solution of a system of equations by *LU* decomposition

Let us have a system of linear equations

$$\mathbf{A}\vec{x} = \vec{b}$$

We replace matrix $\mathbf{A}$ with its *LU* decomposition

$$\mathbf{L}\mathbf{U}\vec{x} = \vec{b}$$

Furthermore, let us denote the product $\mathbf{U}\vec{x} = \vec{y}$. After substitution, we obtain a system of equations

$$\mathbf{L}\vec{y} = \vec{b}$$

This system can be easily solved because $\mathbf{L}$ is a lower triangular matrix. And finally, we can also easily solve the system

$$\mathbf{U}\vec{x} = \vec{y},$$

because $\mathbf{U}$ is an upper triangular matrix.

Solution of a system of equations by *LU* decomposition, example

We have a system of equations

$$\begin{aligned} 2x_1 - x_2 + x_3 &= 1 \\ 4x_1 + x_2 - x_3 &= 5 \\ x_1 + x_2 + x_3 &= 0 \end{aligned}$$

We perform the ***LU*** decomposition of the system matrix $\mathbf{A}$

$$\mathbf{A} = \begin{pmatrix} 2 & -1 & 1 \\ 4 & 1 & -1 \\ 1 & 1 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ \frac{1}{2} & \frac{1}{2} & 1 \end{pmatrix} \begin{pmatrix} 2 & -1 & 1 \\ 0 & 3 & -3 \\ 0 & 0 & 2 \end{pmatrix}$$

Solution of a system of equations by *LU* decomposition, example (cont.)

First, we solve the system $\mathbf{L}\vec{y} = \vec{b}$

$$\begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ \frac{1}{2} & \frac{1}{2} & 1 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix} = \begin{pmatrix} 1 \\ 5 \\ 0 \end{pmatrix}$$

$$\begin{aligned} y_1 &= 1 \\ y_2 &= 5 - 2y_1 = 3 \\ y_3 &= 0 - \frac{1}{2}y_1 - \frac{1}{2}y_2 = -2 \end{aligned}$$

Solution of a system of equations by *LU* decomposition, example (cont.)

Subsequently, we solve the system $\mathbf{U}\vec{x} = \vec{y}$

$$\begin{pmatrix} 2 & -1 & 1 \\ 0 & 3 & -3 \\ 0 & 0 & 2 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 1 \\ 3 \\ -2 \end{pmatrix}$$

$$\begin{aligned} x_3 &= \frac{-2}{2} = -1 \\ x_2 &= \frac{3 - (-3)x_3}{3} = \frac{3 - (-3)(-1)}{3} = 0 \\ x_1 &= \frac{1 - x_3 - (-1)x_2}{2} = \frac{1 - (-1)}{2} = 1 \end{aligned}$$

LU-decomposition of a matrix, notes

- In practice, ***LU***-decomposition is used to solve systems of linear equations.
- Using ***LU***-decomposition, it is possible to efficiently solve multiple systems of equations with the same system matrix.
- The matrices $\mathbf{L}$ and $\mathbf{U}$ can be stored together in one matrix – from the matrix $\mathbf{L}$ we store only the elements below the diagonal. Why?
- If it is necessary to perform partial pivoting in the matrix $\mathbf{A}$, i.e., to swap rows, then the decomposition has the form

  $$\mathbf{PA} = \mathbf{LU}$$

  and from this

  $$\mathbf{A} = \mathbf{P}^{-1}\mathbf{LU},$$

  where $\mathbf{P}$ is a permutation matrix.

Permutation Matrix

- Represents a permutation of n elements as a matrix
- A square binary matrix of order n, with one 1 in each row and column, and the rest 0
- For every permutation matrix $\mathbf{P}$ applies:
  - left multiplication, $\mathbf{PM}$, results in a permutation of the rows of matrix $\mathbf{M}$, where $\mathbf{M}$ is a matrix with n rows
  - right multiplication, $\mathbf{MP}$, results in a permutation of the columns of matrix $\mathbf{M}$, where $\mathbf{M}$ is a matrix with n columns
  - $\mathbf{P}$ is orthogonal, i.e. its inverse matrix is equal to its transpose, $\mathbf{P}^{-1} = \mathbf{P}^{T}$

Permutation matrix, example

$$\pi = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 3 & 2 & 4 & 1 \end{pmatrix} \quad \leftrightarrow \quad R_\pi = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{pmatrix}$$

$$\updownarrow \qquad\qquad\qquad \updownarrow$$

$$C_\pi = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \quad \leftrightarrow \quad \pi^{-1} = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 4 & 2 & 1 & 3 \end{pmatrix}$$

Transform and Conquer

Balanced Search Trees

Binary Search Trees – review

- Fundamental data structure for implementing sets, dictionaries etc.
- Each node contains one key; a total order must be defined over the keys.
- For each node, all keys in the left subtree are smaller than the key in the given node and in the right subtree are all keys greater.
- Average time complexity of search, insertion, and deletion operations is $\Theta(\log_2 n)$.
- Worst-case scenario is however still $\Theta(n)$ – the tree degenerates into a list.

Balanced Search Trees

Possible solution for the worst case:

Proactive Measures

- transformation into a balanced binary tree using rotations
- various definitions of balance
- AVL trees, red-black trees, splay trees.

Representation Change

- multiple keys in one node,
- 2-3 trees, 2-3-4 trees, B-trees.

Authors

- Georgij Maximovič Adelson-Velskij and
- Jevgenij Michajlovič Landis

First published in 1962.

Definition

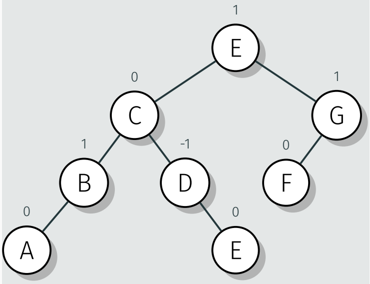
The **balance factor** of a node u is the difference between the heights of its left and right subtrees. The height of an empty tree is defined as -1.

Definition

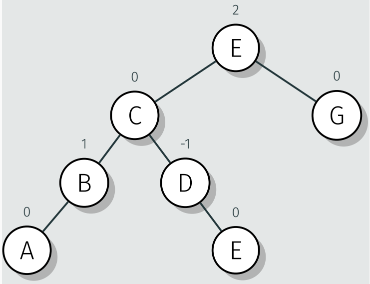
A binary search tree is called an **AVL tree** if and only if the balance factor for each node in the tree is either -1, 0, or +1.

AVL trees – example

AVL tree

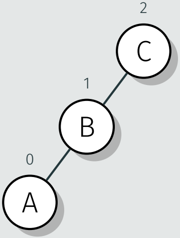


This is not an AVL tree

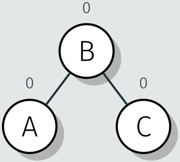


- Insertion of a new node, or deletion of an existing one, can cause imbalance in the AVL tree.
- Balance must be restored after each such operation.
- Balance is restored using **rotations**.
- Rotation is a local transformation of the tree at those nodes where the balance factor reaches a value of -2 or 2.
- If there are multiple such nodes, we always start with the node at the lowest level (closest to the leaves of the tree) and proceed upwards towards the root of the tree.
- There are a total of four rotations – two pairs of mutually mirror-symmetric rotations.

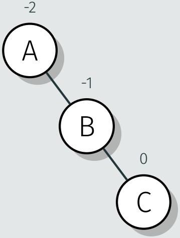
Right rotation



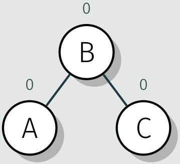
Operation result



Left rotation

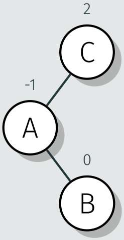


Operation result

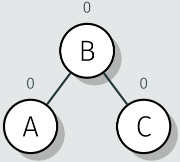


Double rotations

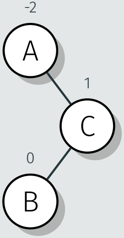
Left-Right rotation



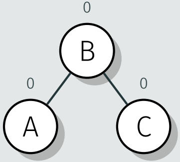
Operation result



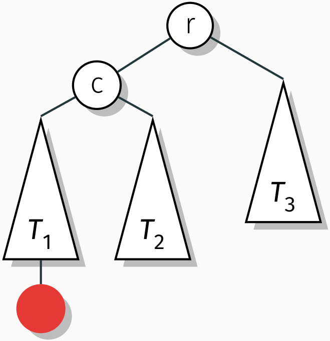
Right-Left rotation



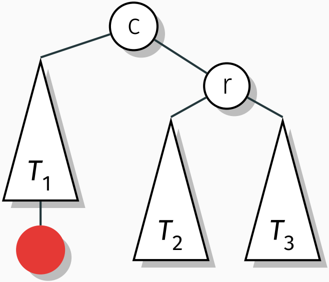
Operation result



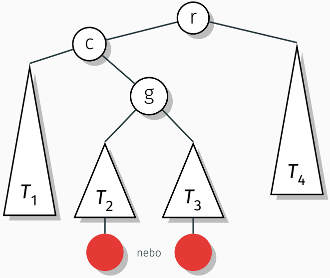
AVL trees – general scheme of right rotation



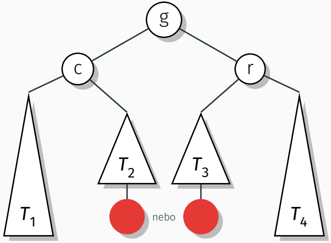
Operation result



AVL trees – general scheme of LR rotation



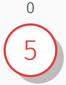
Operation result



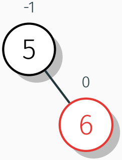
AVL trees – properties of rotations

- Constant time complexity – only pointers between nodes are moved, not data.
- Rotations preserve the ordering of keys in the tree – after completing a rotation, the "left" side always contains smaller keys, the "right" side always contains larger keys.

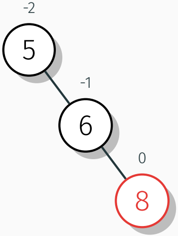
Insertion of 5



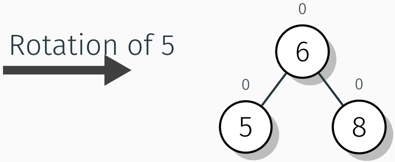
Insertion of 6



Insertion of 8

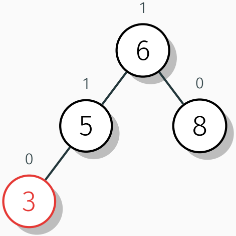


Left Rotation of 5



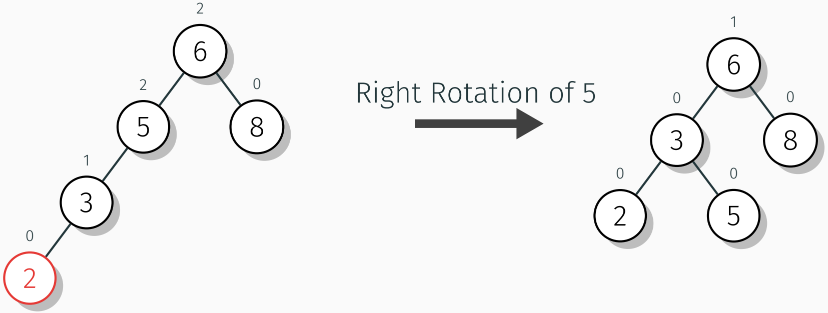
AVL Trees – Sequential Construction of the Tree (cont.)

Insertion of 3



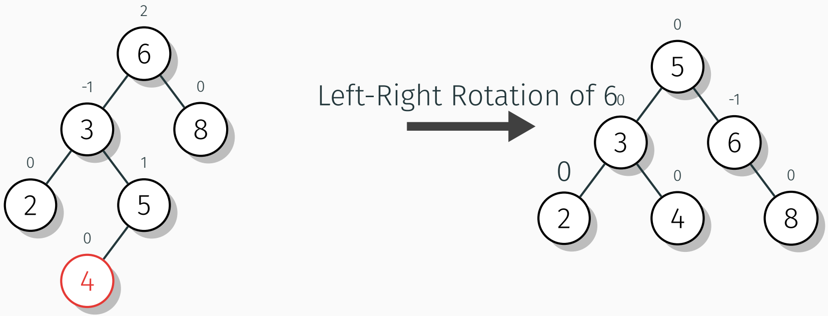
AVL Trees – Sequential Construction of the Tree (cont.)

Insertion of 2



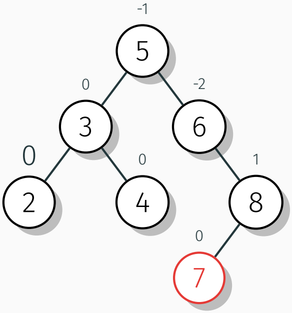
AVL Trees – Sequential Construction of the Tree (cont.)

Insertion of 4

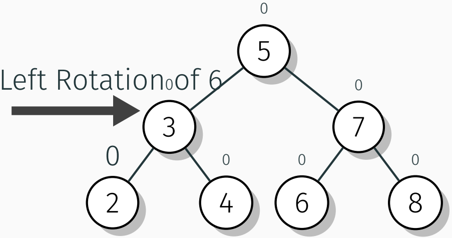


AVL Trees – Sequential Construction of the Tree (cont.)

Insertion of 7



Right-Left Rotation of 6



AVL trees – properties

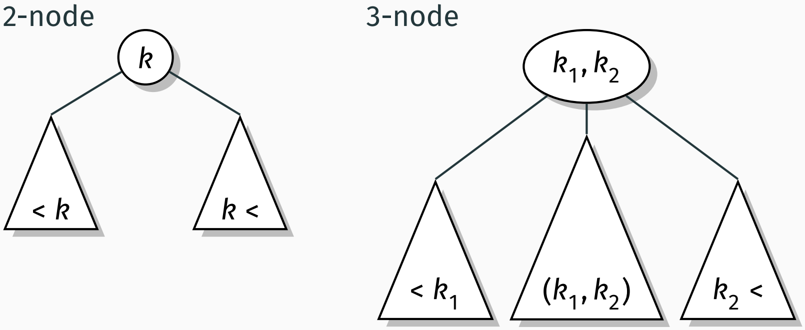
- The height of an AVL tree with n nodes is bounded by

$$\lfloor \log_2 n \rfloor \leq h < 1.4405 \log_2(n + 2) - 1.3277$$

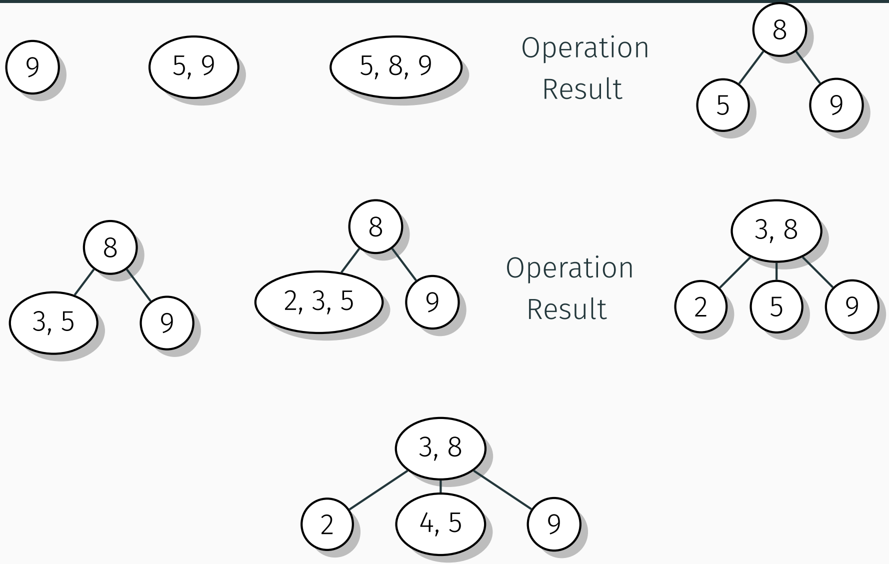
- Search and insertion operations therefore proceed with a complexity of $\Theta(\log_2 n)$ even in the worst case.
- The average height of an AVL tree constructed from a random sequence of n keys is $1.01 \log_2 n + 0.1$.
- Node deletion is more complicated, but still falls within the logarithmic complexity class.
- Disadvantages – a large number of rotations during tree balancing.

2-3 trees

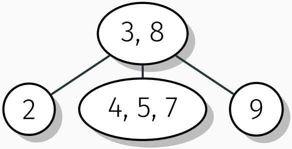
2-3 trees – types of nodes



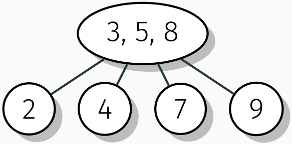
Construction of a 2-3 tree from the sequence 9, 5, 8, 3, 2, 4, 7



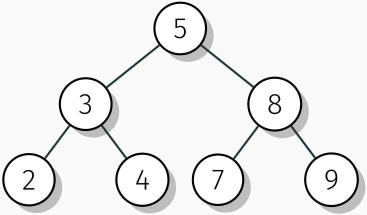
Construction of a 2-3 tree from the sequence 9, 5, 8, 3, 2, 4, 7 (cont.)



Operation
Result



Operation
Result



Sources for independent study

- Book [1], chapter 6.3, pages 218 – 225
- Book [2], chapters 4.4.6, 4.4.7 and 4.4.8, pages 296 – 310

Transform and Conquer

Heap and Heap Sorting

Heap

Heap – a partially sorted data structure, especially suitable for implementing a priority queue.

Priority Queue – a data structure understood as a multiset, where elements are ordered according to **priority** and supporting operations:

- finding the element with the highest priority,
- removing the element with the highest priority and
- inserting a new element into the queue.

Usage of Priority Queue :

- task scheduling in OS
- graph algorithms such as Prim's, Dijkstra's etc.
- heap sorting – **HeapSort**
- and others...

Heap – distinction of terminology

The term heap in computer science is used to denote:

- a data structure and
- a part of the operating memory during program execution.

In further explanation, we will deal with the heap exclusively as a **data structure**.

Heap

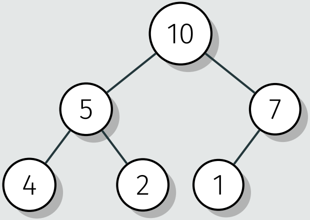
Definition

A **heap** is defined as a binary tree with one key in each node, which satisfies the following two properties:

1. **completeness**, i.e., all levels of the tree are filled, except for the last. In the last level, several leaves may be missing from the right and
2. **parent dominance**, i.e., the key in each node is always greater than or equal to the keys in all its children. In leaves, any key is always considered greater than the keys in non-existent children.

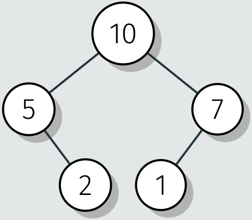
Heap – example

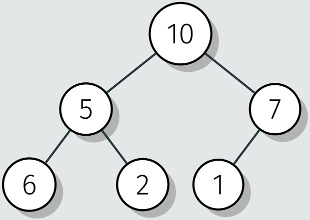
Heap



Not every binary tree is a heap!

These are not heaps – why?





Heap – additional properties

For all heaps, it can be proven that:

1. The keys on each path from the root to a leaf form a **non-increasing** sequence. Otherwise, there are no relationships between the keys, e.g., smaller keys in the left subtree than in the right etc.
2. For n keys, there exists only one complete binary tree. Its height is $\lfloor \log_2 n \rfloor$.
3. The largest key is always at the root of the heap.
4. Each node in the heap is always the root of a heap formed by this node and its descendants.

Heap – array representation

In an array, we store the heap from the root to the leaves and from left to right: Then:

1. internal nodes – the first $\left\lfloor \frac{n}{2} \right\rfloor$, leaves are the remaining $\left\lceil \frac{n}{2} \right\rceil$,
2. the children of a node at position i, where $1 \le i \le \left\lfloor \frac{n}{2} \right\rfloor$, are located at positions $2i$ and $2i + 1$. And conversely, the parent of a node at position j, for $2 \le j \le n$, is located at position $\left\lfloor \frac{j}{2} \right\rfloor$.

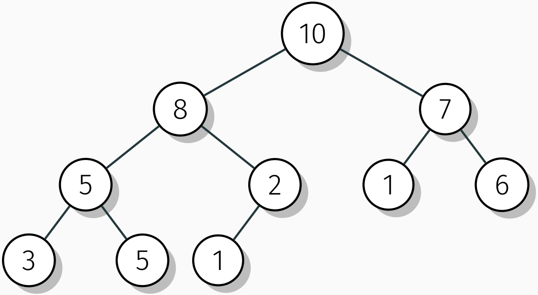
Remark

A heap can be defined as an array $H[1 \dots n]$ in which for each element at index i holds

$$H[i] \ge max\{H[2i], H[2i + 1]\}$$

for all $i = 1, \dots, \left\lfloor \frac{n}{2} \right\rfloor$.

Heap – representation in an array, example



index	1	2	3	4	5	6	7	8	9	10
key	10	8	7	5	2	1	6	3	5	1
	internal nodes					leaves				

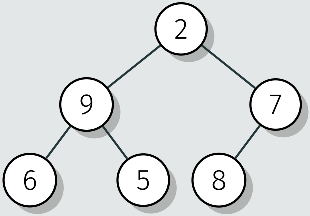
Construction of a heap

A heap can be constructed in two ways:

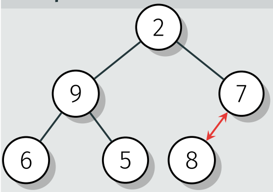
1. **bottom-up** and
2. top-down.

Construction of a heap from the bottom up – example

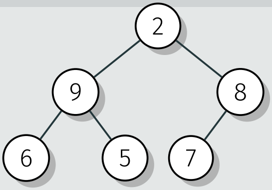
Initial state of the heap



Step 1

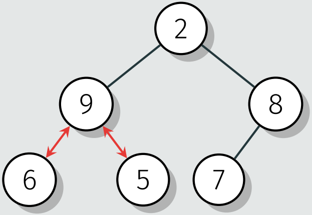


Operation result

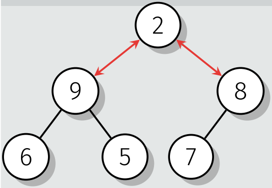


Construction of a heap from the bottom up – example (cont.)

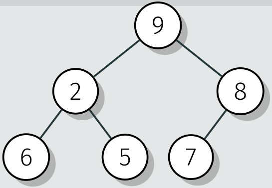
Step 2



Step 3a

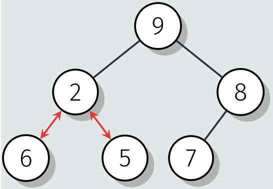


Operation result

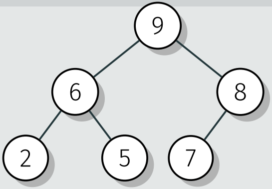


Construction of a heap from the bottom up – example (cont.)

Step 3b



Operation result



Finished heap

Construction of a heap from the bottom up

Input : Array $A[0 \dots n-1]$ with a defined ordering on the array elements, i root of the heap being constructed

Output: Heap with the root at index i

```
procedure Heapify(A, n, i)
    largest ← i;
    l ← 2 ∗ i + 1;
    r ← 2 ∗ i + 2;
    if l < n ∧ A[l] > A[largest] then largest ← l ;
    if r < n ∧ A[r] > A[largest] then largest ← r ;
    if largest ≠ i then
        Swap (A[i], A[largest]);
        Heapify (A, n, largest);
    end
end
```

Construction of a heap from the bottom up

Input : Array $A[0 \ldots n-1]$ with a defined ordering on the array elements

Output: Heap in the array A

```
procedure MakeHeap(A, n)
    for i ← ⌊n/2⌋ − 1 down to 0 do
        Heapify(A, n, i);
    end
end
```

For simplicity, let us assume that $n = 2^k - 1$, i.e., the heap forms a complete binary tree.

The height of the heap is then $h = \lfloor \log_2 n \rfloor$, which can be written as

$$
\begin{aligned}
\lceil \log_2(n+1) \rceil - 1 &= \lceil \log_2(2^k - 1 + 1) \rceil - 1 \\
&= \lceil \log_2(2^k) \rceil - 1 \\
&= k - 1
\end{aligned}
$$

Heap Construction from Bottom to Top – Time Complexity (cont.)

Remark

The expression $\lceil \log_2(n + 1) \rceil$ can be interpreted as the "height of the heap with $n + 1$ elements". We assumed a complete binary tree $\Rightarrow$ the tree with $n + 1$ elements definitely has one more level than the tree with n elements.

Each key from level i will be shifted, in the worst case, to the leaf, i.e., to level h.

Shifting by one level requires two comparisons:

1. finding the larger of both children and
2. testing whether an exchange with the parent is necessary.

Heap Construction from Bottom to Top – Time Complexity (cont.)

The number of comparisons is therefore $2(h-i)$.

The total number of comparisons will be, in the worst case, equal to

$$
\begin{aligned}
C(n) &= \sum_{i=0}^{h-1} \sum_{\text{keys of level } i} 2(h-i) \\
&= \sum_{i=0}^{h-1} 2(h-i)2^i = 2h\sum_{i=0}^{h-1} 2^i - 2\sum_{i=0}^{h-1} i2^i \\
&= 2n - 2\log_2(n+1)
\end{aligned}
$$

Heap Construction from Bottom to Top – Time Complexity (cont.)

Constructing a heap with n elements requires, in the worst case, less than $2n$ comparisons.

Remark

In the derivation, we used the formulas:

$$\sum_{i=0}^{n} 2^i = 2^{n+1} - 1$$

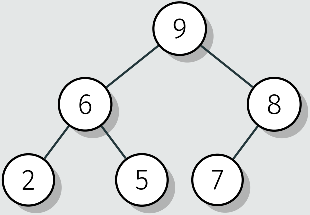
$$\sum_{i=1}^{n} i2^i = 1 \cdot 2 + 2 \cdot 2^2 + \cdots + n2^n = (n-1)2^{n+1} + 2$$

Construction of a heap from top to bottom

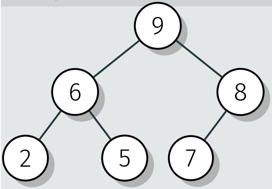
- Repeated insertion of a new key into an existing heap.
  1. We insert the new key at the end of the heap.
  2. We compare the new key with its parent and potentially move the new key up one level.
  3. We continue this process until we encounter a larger parent or reach the root of the heap.
- The height of a heap with n elements is $\approx \log_2 n$, thus the complexity of inserting a key into the heap is $O(\log n)$.
- Construction from top to bottom is therefore more complex than construction from bottom to top.

Construction of a heap from top to bottom – example

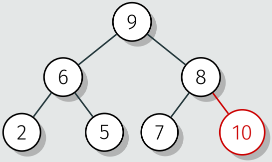
Initial state of the heap



Step 1 – insertion of key 10 at the end of the heap

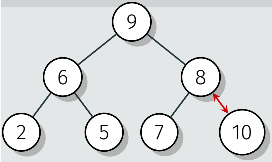


Operation result

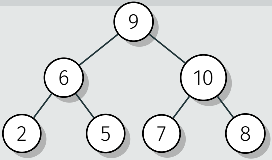


Construction of a heap from top to bottom – example (cont.)

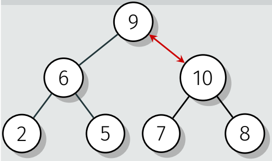
Step 2a – comparison of key 10 with parent



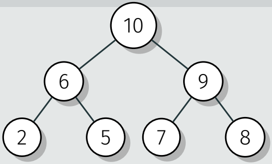
Operation result



Step 2b – comparison of key 10 with parent



Operation result



Removal of the largest key from the heap

Algorithm principle:

1. Swapping the key in the root with the key at the end of the heap.
2. Reducing the heap by one.
3. Heap restoration – testing whether the parent key is greater than the keys in both children and, if necessary, performing a swap. This process is repeated until the parent key is greater than the keys in the children.

Remark

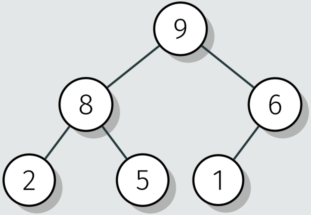
In principle, any key can be removed from the heap. But this operation has no practical significance.

Removal of the largest key from the heap – algorithm complexity

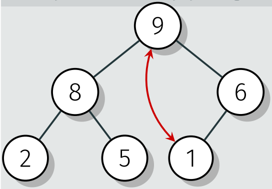
- The number of comparisons necessary to restore the heap is proportional to the height of the heap – we “move” the key from the root down through the levels.
- We always compare the parent with both children – we must find the largest of the given trio.
- The height of the heap is $h \approx \log_2 n$, so the number of comparisons will not be greater than $2h$.
- The complexity of the algorithm is therefore $O(\log n)$.

Removal of the largest key from the heap – example

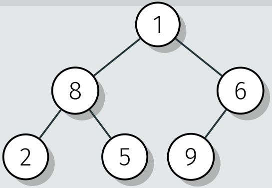
Initial state of the heap



Step 1 – swapping the root with the last element

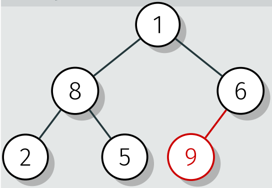


Operation result

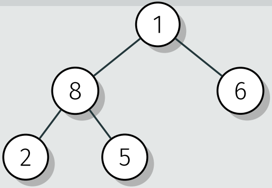


Removal of the largest key from the heap – example (cont.)

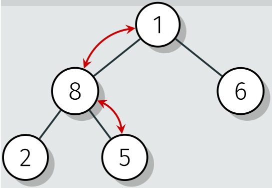
Step 2 – removal of the last node



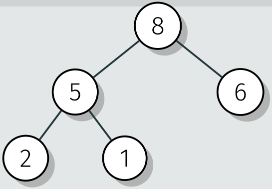
Operation result



Step 3 – heap restoration



Operation result



The algorithm works in two phases:

Heap Construction : for a given array, a heap is constructed.

Removal of Maximum : the algorithm for removing the largest key from the progressively decreasing heap is applied $(n - 1)$ times.

Input : Array $A[0 \ldots n-1]$ with a defined ordering on the array elements

Output: Sorted array A

```
procedure HeapSort(A, n)
    BuildHeap(A, n);
    for i ← n − 1 downto 0 do
        Swap(A[0], A[i]);
        Heapify(A, i, 0);
    end
end
```

Heap sorting – algorithm complexity

- The complexity of the first phase is $O(n)$.
- In the second phase, we progressively remove the largest key from the heap of decreasing size $n, n-1, \dots, 2$. The number of comparisons $C(n)$ is

$$
\begin{aligned}
C(n) &\leq 2\lfloor \log_2(n-1) \rfloor + 2\lfloor \log_2(n-2) \rfloor + \cdots + 2\lfloor \log_2 1 \rfloor \\
&\leq 2\sum_{i=1}^{n-1} \log_2 i \\
&\leq 2\sum_{i=1}^{n-1} \log_2(n-1) = 2(n-1)\log_2(n-1) \leq 2n\log_2 n
\end{aligned}
$$

Thus, $C(n) \in O(n \log n)$.

Heap sorting – algorithm complexity (cont.)

- For both phases, we get $O(n) + O(n \log n) = O(n \log n)$.
- Further complexity analysis can prove that the same complexity applies to the average case as well. Therefore, $\Theta(n \log n)$.
- Heap sorting is comparable to merge sorting.
- However, in practice, it is slower than QuickSort.

Sources for Independent Study

- Book [1], chapter 6.4, pages 226 – 232
- Book [3], chapters 6.1 through 6.4, pages 161 – 172

Transform and Conquer

Horner's Scheme

Value of a Polynomial at a Point

Problem Statement

Given is a polynomial

$$p(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0.$$

Our task is to compute the value of the polynomial $p(x)$ at the point x_0.

Motivation

- Polynomials are used for function approximation, namely
  1. How does a processor calculate the value of the function $\sin(x)$?
  2. Where do the values of the function $\sin(x)$ in mathematical tables come from?

  Using the Taylor series expansion of a function, which is a polynomial!

Taylor expansion of the function $y = f(x)$

The function $f(x)$, which has finite derivatives up to order $n+1$ at point a, can be expressed in the vicinity of point a as an expansion

$$f(x) = f(a) + \frac{f'(a)}{1!}(x-a) + \frac{f''(a)}{2!}(x-a)^2 + \cdots + \frac{f^{(n)}(a)}{n!}(x-a)^n + R_{n+1}^{f,a}(x)$$

For $a = 0$, the expansion is called Maclaurin

$$f(x) = f(0) + \frac{f'(0)}{1!}x + \frac{f''(0)}{2!}x^2 + \cdots + \frac{f^{(n)}(0)}{n!}x^n + R_{n+1}^{f,0}(x)$$

$$\sin(x) = \sin(0) + \frac{\sin'(0)}{1!}x + \frac{\sin''(0)}{2!}x^2 + \cdots + \frac{\sin^{(n)}(0)}{n!}x^n + R_{n+1}^{\sin,0}(x)$$

Derivatives

$$\sin^{(1)} 0 = \cos 0 = 1 \qquad \sin^{(2)} 0 = -\sin 0 = 0$$

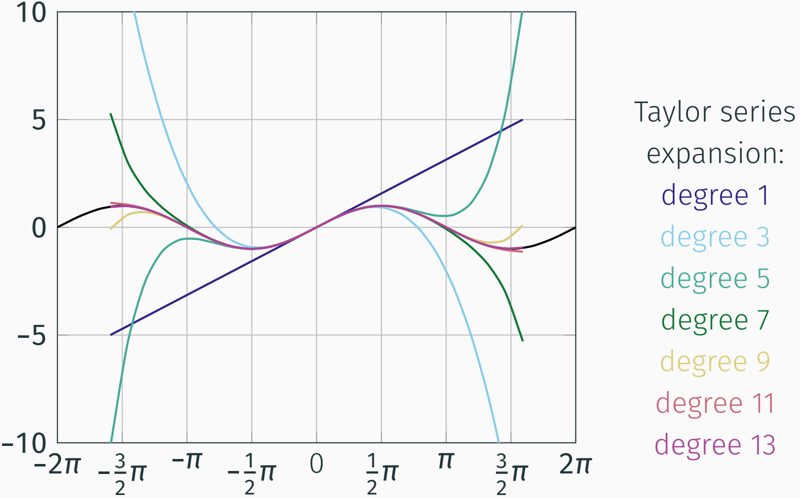
$$\sin^{(3)} 0 = -\cos 0 = -1 \qquad \sin^{(4)} 0 = \sin 0 = 0$$

$$\sin(x) = 0 + \frac{1}{1!}x + \frac{0}{2!}x^2 + \frac{-1}{3!}x^3 + \frac{0}{4!}x^4 + \cdots + R_{n+1}^{\sin,0}(x)$$

Approximation by a 13th-degree polynomial

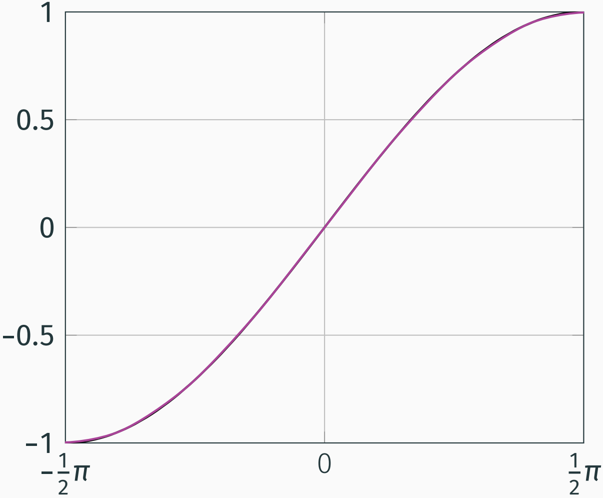
$$\sin(x) \approx x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} - \frac{x^{11}}{11!} + \frac{x^{13}}{13!}$$

Taylor series expansion of the function $y = \sin(x)$ at point 0

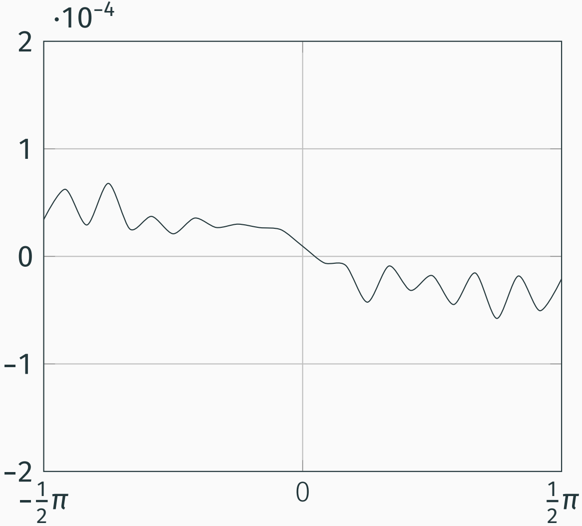


The function $y = \sin(x)$ is displayed in black.

Taylor series expansion of the function $y = \sin(x)$ of degree 13 at point 0



Taylor series expansion of the function $y = \sin(x)$ at point 0, approximation error



Tables of function values

- Using Taylor series expansion, we can approximate the value of the desired function and construct tables.
- Manual calculation – laborious and prone to a vast number of errors.
- Breakthrough idea – numerical computations do not require intelligence! They can be performed **mechanically**!

7.4 cos x (*x* v radiánech)

x	0	1	2	3	4	5	6	7	8	9
0,0	1,0000	1,0000	0,9998	9996	9992	9988	9982	9976	9968	9960
0,1	0,9950	9940	9928	9916	9902	9888	9872	9856	9838	9820
0,2	9801	9780	9759	9737	9713	9689	9664	9638	9611	9582
0,3	9553	9523	9492	9460	9428	9394	9359	9323	9287	9249
0,4	9211	9171	9131	9090	9048	9004	8961	8916	8870	8823
0,5	8776	8727	8678	8628	8577	8525	8473	8419	8365	8309
0,6	8253	8196	8139	8080	8021	7961	7900	7838	7776	7712
0,7	7648	7584	7518	7452	7385	7317	7248	7179	7109	7038
0,8	6967	6895	6822	6749	6675	6600	6524	6448	6372	6294
0,9	6216	6137	6058	5978	5898	5817	5735	5653	5570	5487
1,0	0,5403	5319	5234	5148	5062	4976	4889	4801	4713	4625
1,1	4536	4447	4357	4267	4176	4085	3993	3902	3809	3717
1,2	3624	3530	3436	3342	3248	3153	3058	2963	2867	2771
1,3	2675	2579	2482	2385	2288	2190	2092	1994	1896	1798
1,4	1700	1601	1502	1403	1304	1205	1106	1006	0907	0807
1,5	0707	0608	0508	0408	0308	0208	0108	0008		
1,5									−0,0092	−0,0192
1,6	−0,0292	0392	0492	0592	0691	0791	0891	0990	1090	1189
1,7	−0,1288	1388	1486	1585	1684	1782	1881	1979	2077	2175
1,8	−0,2272	2369	2466	2563	2660	2756	2852	2948	3043	3138
1,9	−0,3233	3327	3421	3515	3609	3702	3795	3887	3979	4070
2,0	−0,4161	4252	4342	4432	4522	4611	4699	4787	4875	4962
2,1	−0,5048	5135	5220	5305	5390	5474	5557	5640	5722	5804
2,2	−0,5885	5966	6046	6125	6204	6282	6359	6436	6512	6588
2,3	−0,6663	6737	6811	6883	6956	7027	7098	7168	7237	7306
2,4	−0,7374	7441	7508	7573	7638	7702	7766	7828	7890	7951
2,5	−0,8011	8071	8130	8187	8244	8301	8356	8410	8464	8517
2,6	−0,8569	8620	8670	8720	8768	8816	8863	8908	8953	8998
2,7	−0,9041	9083	9124	9165	9204	9243	9281	9318	9353	9388
2,8	−0,9422	9455	9487	9519	9549	9578	9606	9633	9660	9685
2,9	−0,9710	9733	9755	9777	9797	9817	9836	9853	9870	9885
3,0	−0,9900	9914	9926	9938	9948	9958	9967	9974	9981	9987
3,1	−0,9991	9995	9998	9999	−1,0000	−1,0000				

k	1	2	3	4	5	6	7	8	9	10
$k \cdot 2\pi$	6,283	12,566	18,850	25,133	31,416	37,699	43,982	50,265	56,549	62,832

$\cos(-x) = \cos x$ $\cos x = \cos(x + k \cdot 2\pi),\ k \in Z$

87

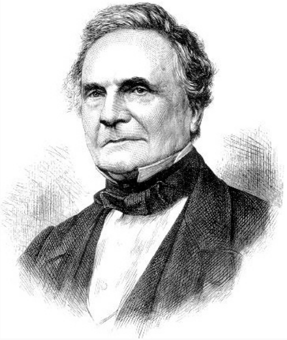
sin *x*, tg *x*

Difference Engine

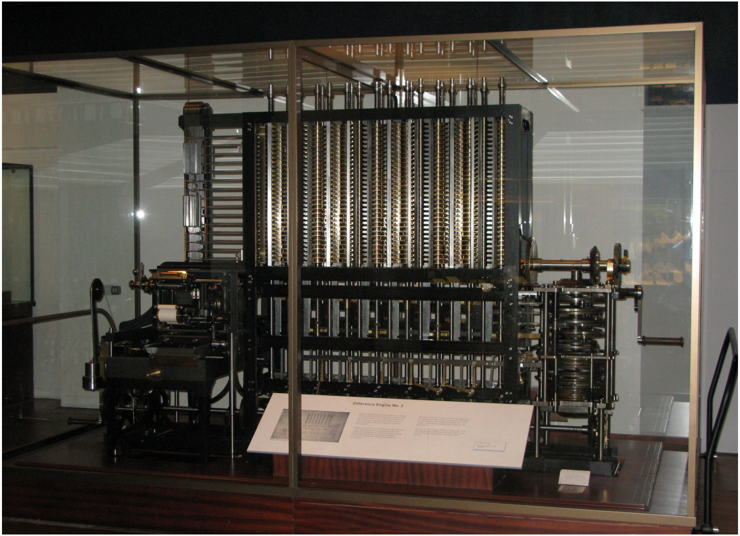
- first programmable computer in the world
- 1819 – commencement of work
- 1822 – prototype completed
- 1823 – work begun on large machine
- 1833 – work halted
- 1842 – government support terminated, 17 thousand pounds spent on project, machine never completed
- 1991 – functional replica!

Charles Babbage

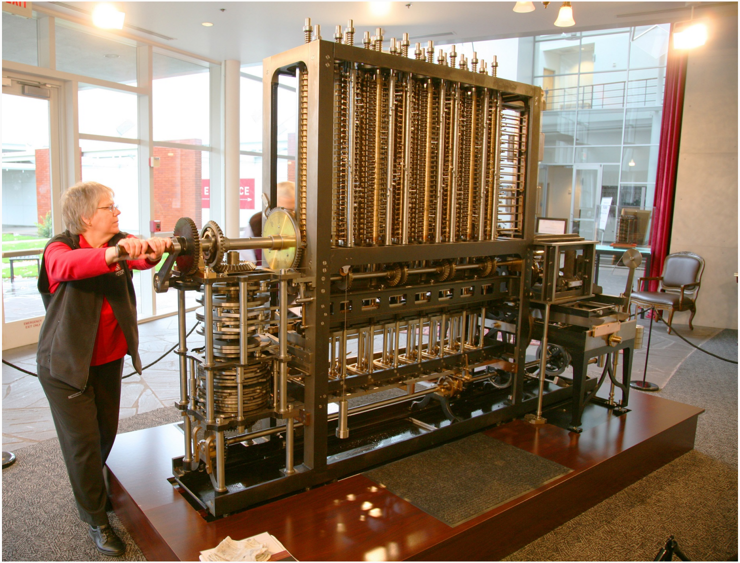
1791 – 1871



Difference Engine



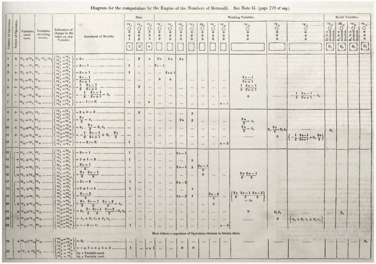
Difference Engine



First programmer in the world?!

Augusta Ada King, Countess of Lovelace (1815 – 1852)

Programmer of the **Analytical Engine**, (Babbage 1837), which was the first general-purpose Turing-complete computer.



Horner's scheme – transformation

Basic idea:

- transformation of a polynomial into another form,
- we gradually extract the variable x from parts of the polynomial.

$$\begin{aligned}
p(x) &= a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1} + a_n x^n \\
&= a_0 + x\left(a_1 + a_2 x + \cdots + a_{n-1} x^{n-2} + a_n x^{n-1}\right) \\
&= a_0 + x\left(a_1 + x\left(a_2 + \cdots + a_{n-1} x^{n-3} + a_n x^{n-2}\right)\right) \\
&\vdots \\
&= a_0 + x\Big(a_1 + x\Big(a_2 + \cdots + x\big(a_{n-1} + a_n x\big)\ldots\Big)\Big)
\end{aligned}$$

It is easy to see that this equality holds by successive multiplication of all parentheses.

Horner's scheme – computation

The value of $p(x_0)$ is computed "from the inside" of the parentheses, progressively calculating the values of b_i

$$\begin{aligned} b_n &= a_n \\ b_{n-1} &= a_{n-1} + b_n x_0 \\ b_{n-2} &= a_{n-2} + b_{n-1} x_0 \\ &\vdots \\ b_0 &= a_0 + b_1 x_0 \end{aligned}$$

The value of b_0 is then equal to $p(x_0)$, since

$$p(x_0) = a_0 + x_0\Big(a_1 + x_0\big(a_2 + \cdots + x_0(a_{n-1} + a_n x_0)\ldots\big)\Big)$$

Horner's scheme – computation (cont.)

and by progressively substituting b_i, we obtain

$$
\begin{aligned}
p(x_0) &= a_0 + x_0\Big(a_1 + x_0\big(a_2 + \cdots + x_0(a_{n-1} + b_n x_0)\ldots\big)\Big) \\
p(x_0) &= a_0 + x_0\Big(a_1 + x_0\big(a_2 + \cdots + x_0(b_{n-1})\ldots\big)\Big) \\
p(x_0) &= a_0 + x_0(b_1) \\
p(x_0) &= b_0
\end{aligned}
$$

Horner's scheme – manual calculation

Calculate the value of the polynomial $p(x) = 2x^3 - 6x^2 + 2x - 1$ at the point $x_0 = 3$.

x_0	x^3	x^2	x^1	x^0
3	2	-6	2	-1
		6	0	6
	2	0	2	5

Standard calculation

$$
\begin{aligned}
p(3) &= 2 \times 3^3 - 6 \times 3^2 + 2 \times 3 - 1 \\
&= 2 \times 27 - 6 \times 9 + 2 \times 3 - 1 \\
&= 54 - 54 + 6 - 1 = 5
\end{aligned}
$$

ALGORITHM *Horner*$(P[0..n], x)$

//Evaluates a polynomial at a given point by Horner's rule
//Input: An array $P[0..n]$ of coefficients of a polynomial of degree n,
// stored from the lowest to the highest and a number x
//Output: The value of the polynomial at x
$p \leftarrow P[n]$
for $i \leftarrow n - 1$ **downto** 0 **do**
    $p \leftarrow x * p + P[i]$
return p

It is clear that the number of multiplications $M(n)$ and the number of additions $A(n)$ equals

$$M(n) = A(n) = \sum_{i=0}^{n-1} 1 = n \in \Theta(n)$$

Computation by brute force

Just for computing $a_n x^n$, the following is needed:

- $n - 1$ multiplications to compute the power
- 1 multiplication to multiply by a_n.

For the same number of multiplications, Horner's algorithm can also compute the remaining $n - 1$ terms of the polynomial!!!

Sources for independent study

- Book [1], chapter 6.5, pages 234 – 239
- Book [3], chapter 30.1, pages 879 – 880

Transform and Conquer

Problem Reduction

Problem Reduction

The purpose of reduction is to transform the problem being solved into another problem that we know how to solve.

Reduction Procedure

1. **Problem 1** – what we want to solve
2. Reduction of **Problem 1** to **Problem 2**
3. **Problem 2** – solvable by algorithm A
4. Execution of algorithm A
5. **Solution to Problem 2**

Least Common Multiple

The least common multiple $lcm(m,n)$ of two natural numbers m and n is defined as the smallest natural number that is divisible by both m and n.

Solution using Prime Factorization

$$
\begin{aligned}
24 &= 2^3 \cdot 3^1 \\
60 &= 2^2 \cdot 3^1 \cdot 5^1 \\
\mathrm{lcm}(24, 60) &= 2^3 \cdot 3^1 \cdot 5^1 = 120
\end{aligned}
$$

Solution using Greatest Common Divisor

It can be proven that

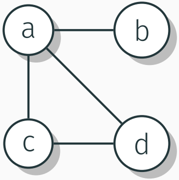
$$\mathrm{lcm}(m,n) = \frac{mn}{\mathrm{gcd}(m,n)}$$

$\mathrm{gcd}(m,n)$ can be computed efficiently using the Euclidean algorithm.

Number of walks in a graph

Problem statement: Calculate the number of walks between pairs of vertices in a given graph G.

Solution: It can be proven that the number of different walks of length k between vertices i and j is equal to the element a_{ij} of the matrix $\mathbf{A}^k$, where A is the adjacency matrix of graph G.



$$\mathbf{A} = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{pmatrix} \end{array} \quad \mathbf{A}^2 = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{pmatrix} 3 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 2 & 1 \\ 1 & 1 & 1 & 2 \end{pmatrix} \end{array}$$

From a to a, there are three walks of length 2: $a - b - a$, $a - c - a$, $a - d - a$

From a to c, there is one walk of length 2: $a - d - c$

Maximization Problem – finding the maximum of function $f(x)$

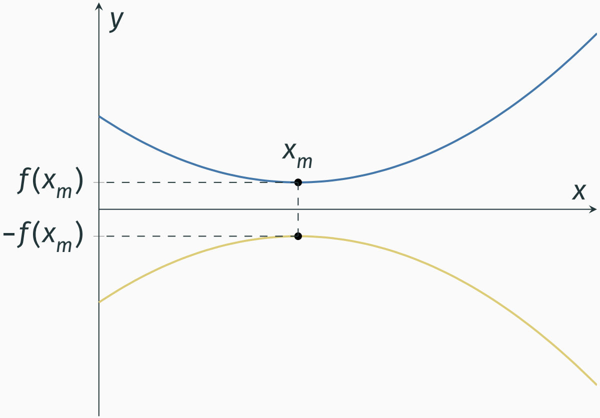
Minimization Problem – finding the minimum of function $f(x)$

How to Solve the Situation?

- We need to minimize function $f(x)$, but
- we only have a maximization algorithm available.

Can we use a maximization algorithm for a minimization problem? Or vice versa?

Reduction of Optimization Problems



$$\min f(x) = -\max\left[-f(x)\right]$$
$$\max f(x) = -\min\left[-f(x)\right]$$

Goat, wolf and cabbage

- On the riverbank, there is a ferryman, a goat, a wolf, and cabbage.
- The ferryman must transport the goat, the wolf, and the cabbage to the other bank using a boat.
- The boat can hold at most one of the entities being transported, in addition to the ferryman.
- On the same bank, the pairs goat and cabbage and wolf and goat cannot be left together without the ferryman's supervision.
- The task is to devise a transportation plan or prove that no solution exists.

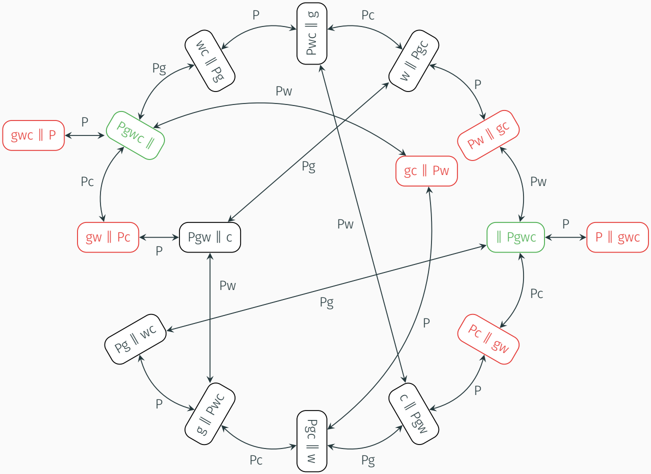
The oldest written form of the problem dates back to the 9th century...

Goat, wolf and cabbage – state space

State – represents the occupancy of both riverbanks, e.g. Gw||c

Transition between states – path from one riverbank to the other, with possible transportation

Goat, wolf and cabbage – state space graph



Solution to the problem – finding a directed path from the initial state to the final state through breadth-first traversal.

Sources for Independent Study

- Book [1], chapter 6.6, pages 240 – 248

Thanks for your attention